Safetensors vs GGUF

Jun 24, 2024

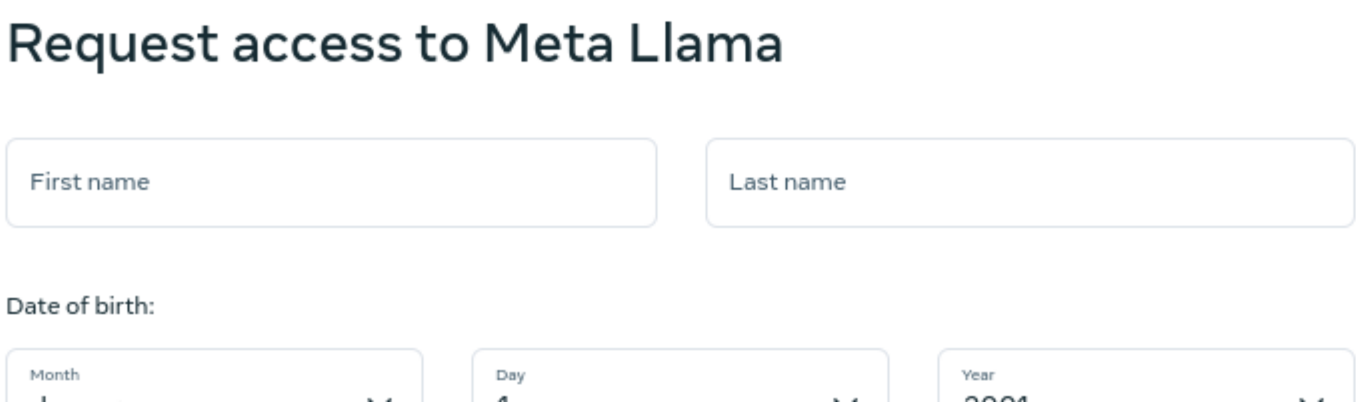
There are two popular formats found in the wild when getting a Llama 3 model: .safetensors and .gguf extension. Let's get Llama 3 with both formats, analyze them, and perform inference on it (generate some text with it) using the most popular library for each format, covering:

- Getting Llama 3 from Meta website
- Converting .pth to .safetensors
- Safetensors origins
- Safetensors data format
- Safetensors inference (with HF's transformers)
- Converting .safetensors to .gguf
- GGUF origins
- GGUF data format
- GGUF inference (with llama.cpp)
- Which one to pick?

Note that the first two sections of this article can be skipped by downloading the [.safetensors model from Hugging Face].

Getting Llama 3 from Meta Website

Let's first download the model directly from [Meta website]. There's a [download.sh] that we have to run while providing to it a pre-signed URL that we only get after filling out a form:



Request access to Meta Llama

First name

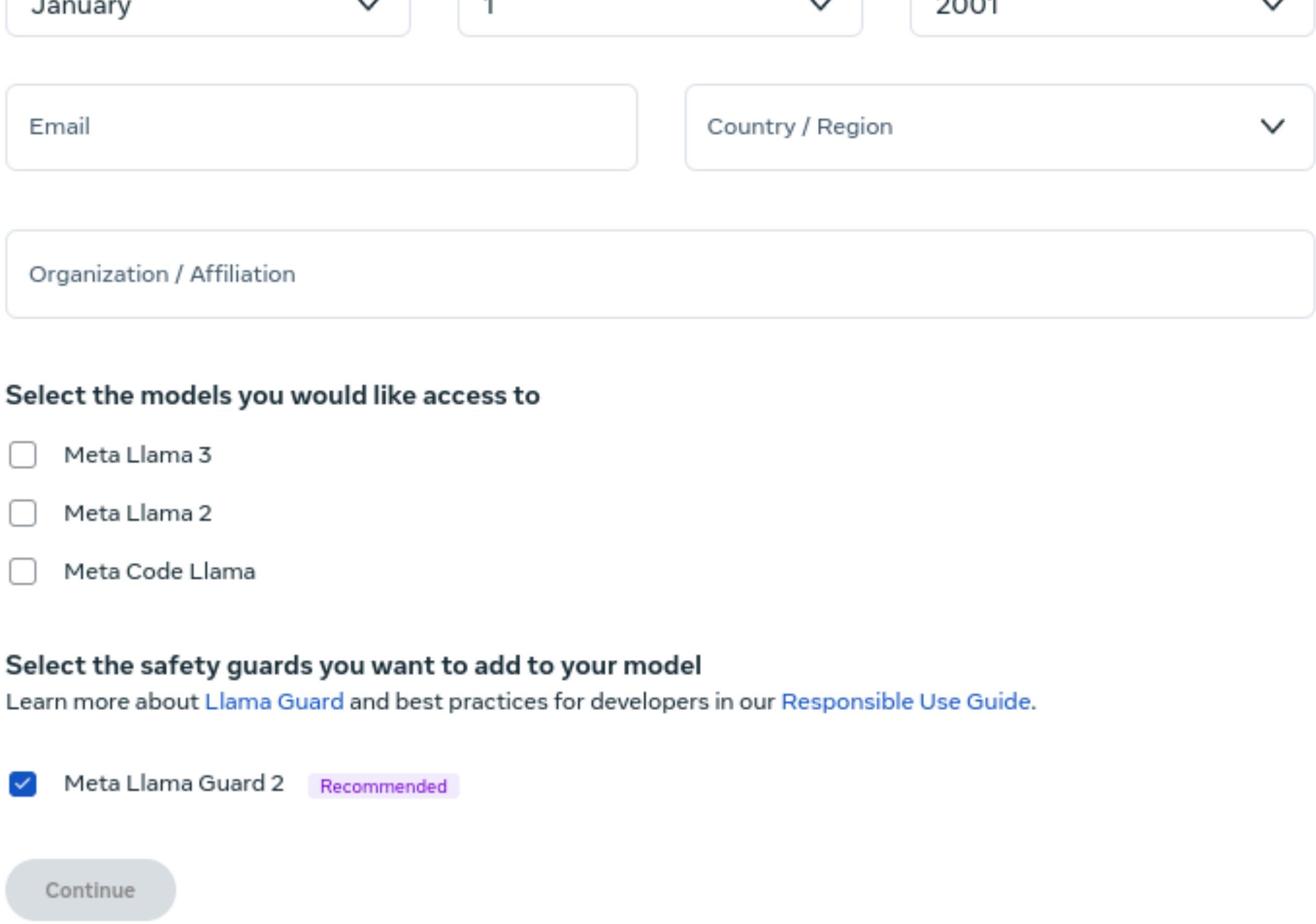
Last name

Date of birth:

Month

Day

Year



It's not as bad as it sounds: it does not require a Facebook login and the download script doesn't do much other than executing some wget and performing checksum. I've ended up with a **"consolidated.00.pth"** file after this process. It's ZIP file, let's see its contents:

```
consolidated.00/
├── data
│   ├── 0
│   ├── 1
│   ├── 2
│   ├── 3
│   ├── 4
│   ├── ...
│   └── 290
├── byteorder
├── data.pkl
└── version
```

Piggybacking on [PyTorch documentation](), let's go file by file:

- **data/**: this is *probably* the **torch.Storage** objects of the model, filtered out from data.pkl (see below)
- **byteorder**: mine just says "little" for little endianness (I wonder if there builds for big-endian and if folks are running this on unusual processors)
- **data.pkl**: the pickled object passed to the save function (the model) excluding any **torch.Storage** object
- **version**: mine just says "3", the documentation is not clear about this file but I'm assuming this is the model version ("3" for "Llama 3")

Converting .pth to .safetensors

There's a convert_llama_weights_to_hf.py script on Hugging Face's Transformers repository. Usage options:

```
-h, --help
show this help message and exit

--input_dir INPUT_DIR
Location of LLaMA weights, which contains tokenizer.model and model folders

--model_size {7B,8B,8Bf,7Bf,13B,13Bf,30B,34B,65B,70B,70Bf,tokenizer_only}
'f' models correspond to the finetuned versions, and are specific to the Llama2 official release. For more

--output_dir OUTPUT_DIR
Location to write HF model and tokenizer

--safe_serialization SAFE_SERIALIZATION
Whether or not to save using `safetensors`.

--llama_version {1,2,3}
Version of the Llama model to convert. Currently supports Llama1 and Llama2. Controls the context size
```

Full commands, including installing required packages with pip:

```
$ python3 -m venv venv

$ source venv/bin/activate

(venv) $ pip install accelerate blobfile tiktoken transformers tokenizers torch
Collecting accelerate
  Using cached accelerate-0.31.0-py3-none-any.whl (309 kB)
Collecting blobfile
  Using cached...

(venv) $ python3 venv/lib/python3.10/site-packages/transformers/models/llama/convert_llama_weights_to_hf.py
Saving a LlamaTokenizerFast to ..
Fetching all parameters from the checkpoint at Meta-Llama-3-8B/.
...
```

After this I've got the following files in the folder where I've executed the script (just like the list of files we see at Hugging Face):

```
config.json
generation_config.json
model-00001-of-00004.safetensors
model-00002-of-00004.safetensors
model-00003-of-00004.safetensors
model-00004-of-00004.safetensors
model.safetensors.index.json
special_tokens_map.json
tokenizer_config.json
```

```
tokenizer.json
```

Safetensors Origins

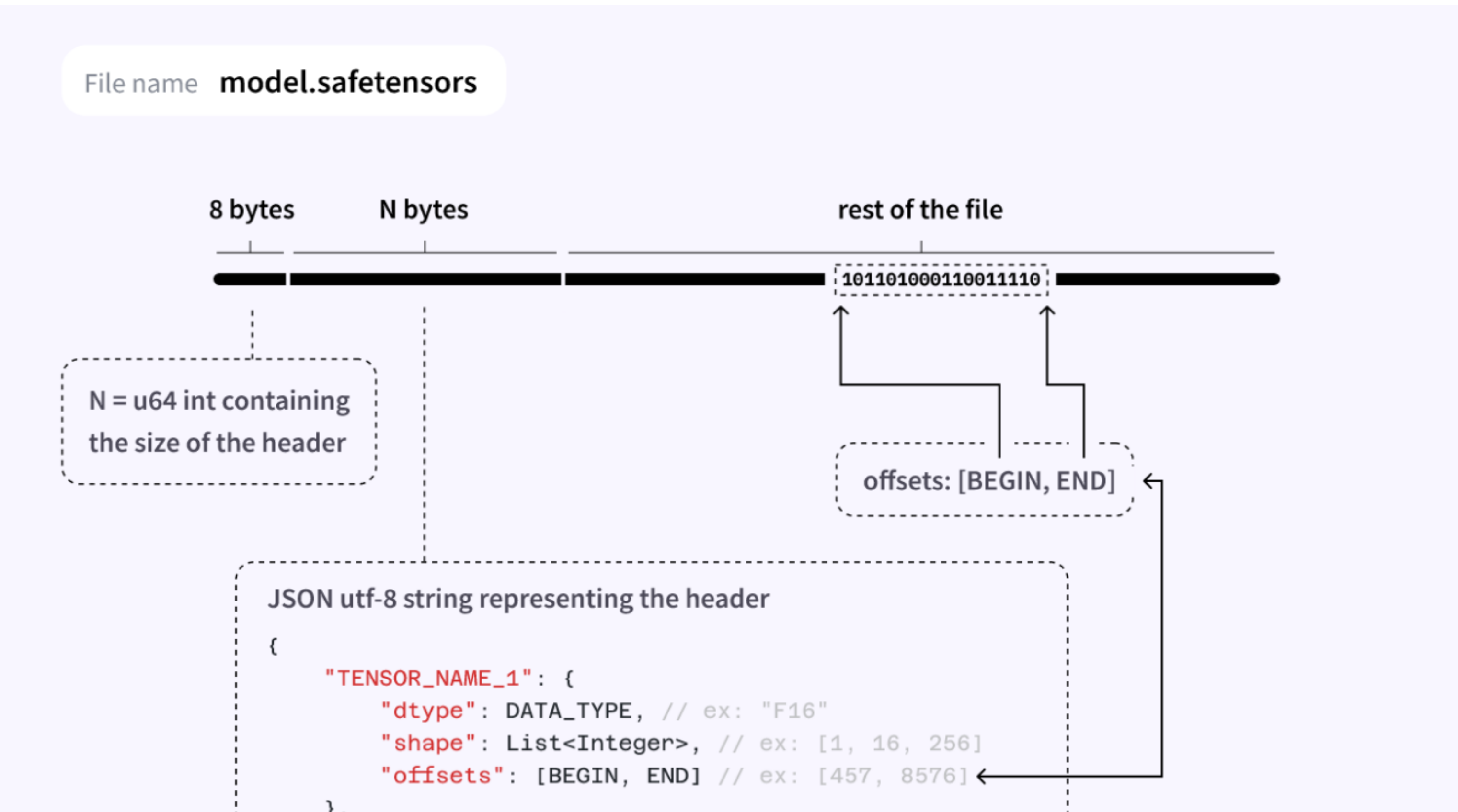
Many in the machine learning community use pickle to save models, but it's a known insecure way to share data since it's easy to execute arbitrary code. Here's an example:

```
import pickle
import os

# Create a malicious class
class Malicious:
    def reduce(self):
        # os.system will execute the command
        return (os.system, ('echo "This is malicious code!"',))

# Serialize the malicious object
malicious_data = pickle.dumps(Malicious())

# Deserialize the malicious object (this will execute the command)
pickle.loads(malicious_data)
```

This was the main intent of this new format, hence 'safe' in the name. But it not only addresses this security issue but also improves the loading time of tensors by using CUDA to transfer the tensor directly to the GPU and skipping unnecessary CPU allocations (source).

Safetensors Data Format

The data format of the .safetensors is somewhat simple: a header size at the beginning of the file followed by a JSON object (of such size):



```
        "TENSOR_NAME_2": {...},
            ...,
        "__metadata__": {...} // special key for storing
                                 free form text-to-text map
    }
    // DATA_TYPE can be one of ["F64","F32","F16","BF16",
    "I64","I32","I16","I8","U8","BOOL"]
```

With Python, let's read the header of the **"model-00001-of-00004.safetensors"** that we got from the previous section:

```python
from struct import unpack
from json import dumps, loads

with open('model-00001-of-00004.safetensors', mode='rb') as f:

    # reading first 8 bytes to get the size of the header
    size_bytes = f.read(8)
    size, = unpack('<Q', size_bytes)

    # the header is 9512 bytes long, reading it and printing a pretty JSON
    header_bytes = f.read(size)
    header_json = loads(header_bytes)
    header_pretty = dumps(header_json, indent=4)

    print(header_pretty)
```

Here is the result (shortened here, the result is 889 lines long):


```json
{
    "__metadata__": {
        "format": "pt"
    },
    "model.embed_tokens.weight": {
        "dtype": "BF16",
        "shape": [
            128256,
            4096
        ],
        "data_offsets": [
            0,
            1050673152
        ]
    },
    "model.layers.0.input_layernorm.weight": {
        "dtype": "BF16",
        "shape": [
            4096
        ],
        "data_offsets": [
            1050673152,
            1050681344
        ]
    },
```

```
    "model.layers.0.mlp.down_proj.weight": {
        "dtype": "BF16",
        "shape": [
            4096,
            14336
        ],
        "data_offsets": [
            1050681344,
            1168121856
        ]
    },
    ...
}
```

The **“pt”** format *probably* stands for “PyTorch” and we got multiple inner objects per layer as expected.

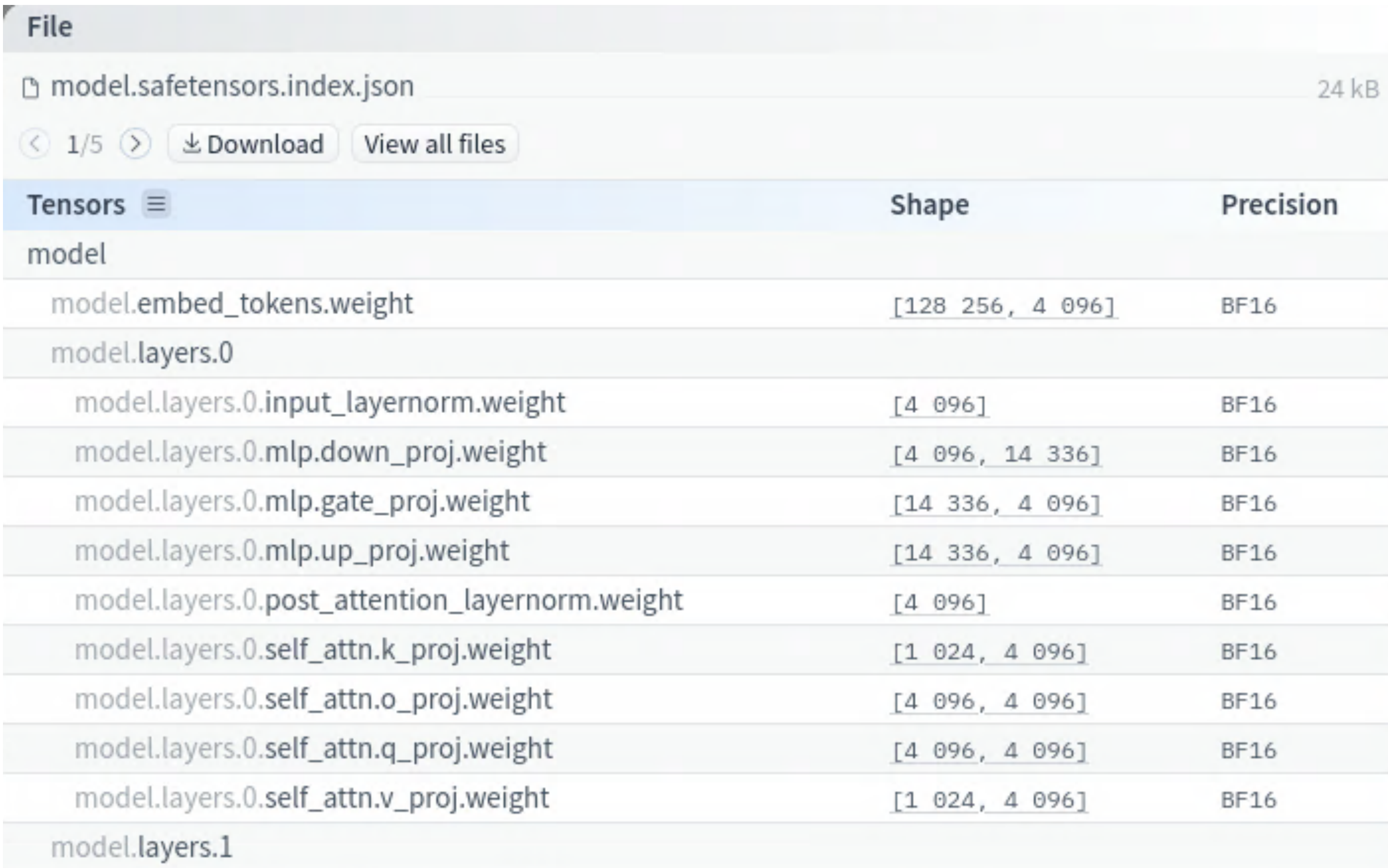
On each layer, we got **“BF16”** standing for bfloat16, which apparently is a way to save space (16-bit instead of 32-bit) while easing the conversion to traditional 32-bit when compared to a **“F16”** (see here).

“shape” is the size of the layers (how many parameters). The first layer of the example has 525,336,576 parameters (128,256 x 4,096).

“data_offsets” represents the starting byte and ending byte of the layer data stored in the model file.

We can also see the header information directly from Hugging Face website interface by clicking on this button:

Which opens:


File
model.safetensors.index.json 24 kB
1/5 Download View all files

Tensors	Shape	Precision
model		
model.embed_tokens.weight	[128 256, 4 096]	BF16
model.layers.0		
model.layers.0.input_layernorm.weight	[4 096]	BF16
model.layers.0.mlp.down_proj.weight	[4 096, 14 336]	BF16
model.layers.0.mlp.gate_proj.weight	[14 336, 4 096]	BF16
model.layers.0.mlp.up_proj.weight	[14 336, 4 096]	BF16
model.layers.0.post_attention_layernorm.weight	[4 096]	BF16
model.layers.0.self_attn.k_proj.weight	[1 024, 4 096]	BF16
model.layers.0.self_attn.o_proj.weight	[4 096, 4 096]	BF16
model.layers.0.self_attn.q_proj.weight	[4 096, 4 096]	BF16
model.layers.0.self_attn.v_proj.weight	[1 024, 4 096]	BF16
model.layers.1		

model.layers.1.input_layernorm.weight	[4 096]	BF16
model.layers.1.mlp.down_proj.weight	[4 096, 14 336]	BF16
model.layers.1.mlp.gate_proj.weight	[14 336, 4 096]	BF16
model.layers.1.mlp.up_proj.weight	[14 336, 4 096]	BF16

Safetensors Inference (with HF's transformers)

The transformers library makes it easy:

```
from torch import bfloat16
from transformers import pipeline

chat = [
    {'role': 'system', 'content': 'AMD vs NVIDIA is an interesting topic'},
]

pipe = pipeline('text-generation', '/path/to/safetensors/model/Meta-Llama-3-8B/', torch_dtype=bfloat16, dev
response = pipe(chat, max_new_tokens=250)
print(response[0]['generated_text'][-1]['content'])
```

Weirdly, when I set **"device_map='auto'"** (and installing **"accelerate"** package) it was supposed to automatically distribute the model in RAM and VRAM accordingly, but it didn't quite work out:

```
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 1002.00 MiB. GPU
```

I had to offload the generation to my CPU (by setting **"device='cpu'"** instead) since my RTX 2060 has only 6GB of VRAM. This made the execution super slow… but it worked:

```
$ python3 -m venv venv

$ source venv/bin/activate

(venv) $ pip install torch transformers

(venv) $ python3 test.py
Loading checkpoint shards: 100%|██████████| 4/4 [00:03<00:00,  1.07it/s]
WARNING:root:Some parameters are on the meta device device because they were offloaded to the cpu.
Special tokens have been added in the vocabulary, make sure the associated word embeddings are fine-tuned c
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.

## AMD vs NVIDIA: The Pros and Cons

AMD vs NVIDIA is a topic that has been debated for years. Both companies have their pros and cons, and it c

## AMD vs NVIDIA: The Pros

AMD is a company that has been around for many years, and they have a lot of experience in the industry. Th

...
```

Converting .safetensors to .gguf

There's a convert-hf-to-gguf-update.py script on llama.cpp's GitHub repository. Usage options:

```
-h, --help
show this help message and exit

--vocab-only
extract only the vocab

--awq-path AWQ_PATH
Path to scale awq cache file

--outfile OUTFILE
path to write to; default: based on input. {ftype} will be replaced by the outtype.

--outtype {f32,f16,bf16,q8_0,auto}
output format - use f32 for float32, f16 for float16, bf16 for bfloat16, q8_0 for Q8_0, auto for the highes
16-bit float type depending on the first loaded tensor type

--bigendian
model is executed on big endian machine

--use-temp-file
use the tempfile library while processing (helpful when running out of memory, process killed)

--no-lazy
use more RAM by computing all outputs before writing (use in case lazy evaluation is broken)

--model-name MODEL_NAME
name of the model

--verbose
increase output verbosity
```

Before running the script, let's clone the repository and move the conversion script to the folder where "gguf" python module is located:

```
$ git clone https://github.com/ggerganov/llama.cpp.git
Cloning into 'llama.cpp'...
remote: Enumerating objects: 27710, done.
...

$ cd llama.cpp

$ mv convert-hf-to-gguf.py gguf-py

$ cd gguf-py
```

Keeping things simple, I decided to leave the options to their defaults and just run it (installing the necessary packages first):

```
$ python3 -m venv venv

$ source venv/bin/activate

(venv) $ pip install numpy sentencepiece torch transformers
Collecting numpy
  Using cached numpy-2.0.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (19.3 MB)
Collecting sentencepiece
  Using cached...

(venv) $ python3 convert-hf-to-gguf-update.py "/path/to/safetensors/model/Meta-Llama-3-8B/" --outfile "Meta
INFO:hf-to-gguf:Loading model: ..
INFO:gguf.gguf_writer:gguf: This GGUF file is for Little Endian only
...
```

Unfortunately, I had to use “fp16” for the conversion instead of “bf16” as the original model due to what seems to be a bug. There is a pull request open that I believe addresses this (#7843).

GGUF Origins

First, we have to introduce GGML.

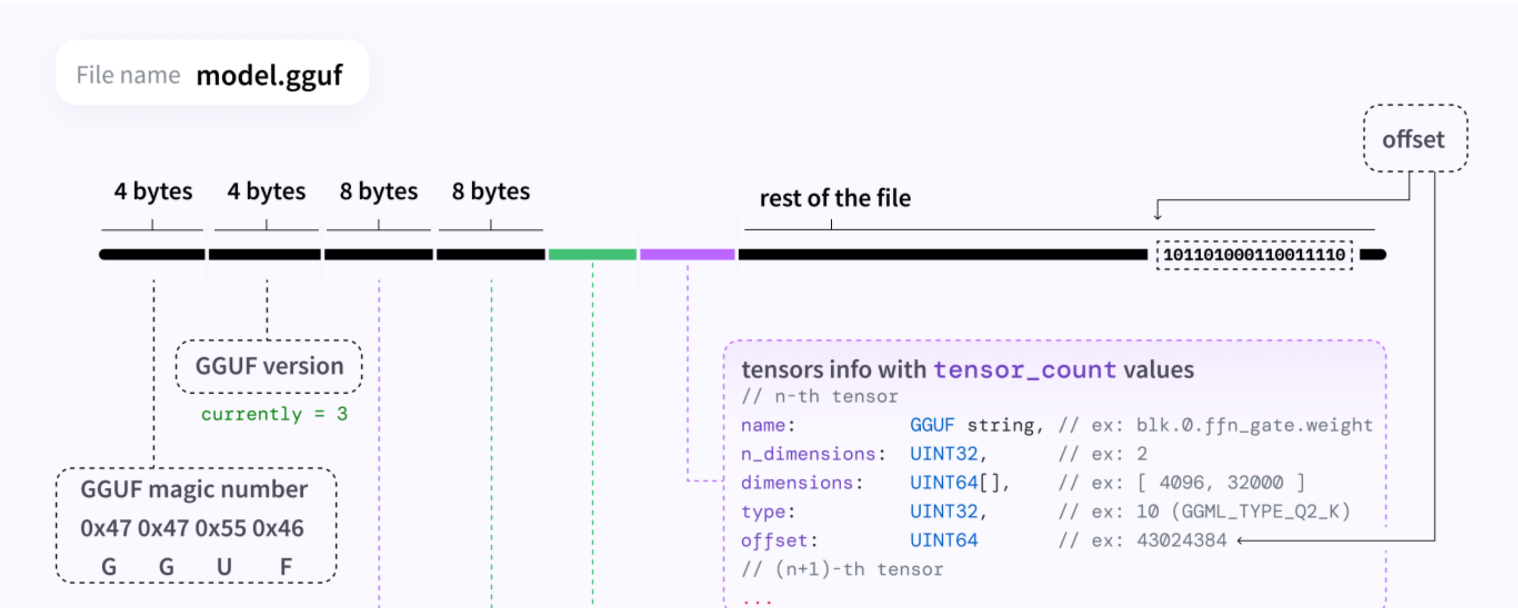
GGML (“GG” refers to the initials of its author, Georgi Gerganov), is a C library that helps manipulate tensors, specifically when performing inference on large language models.

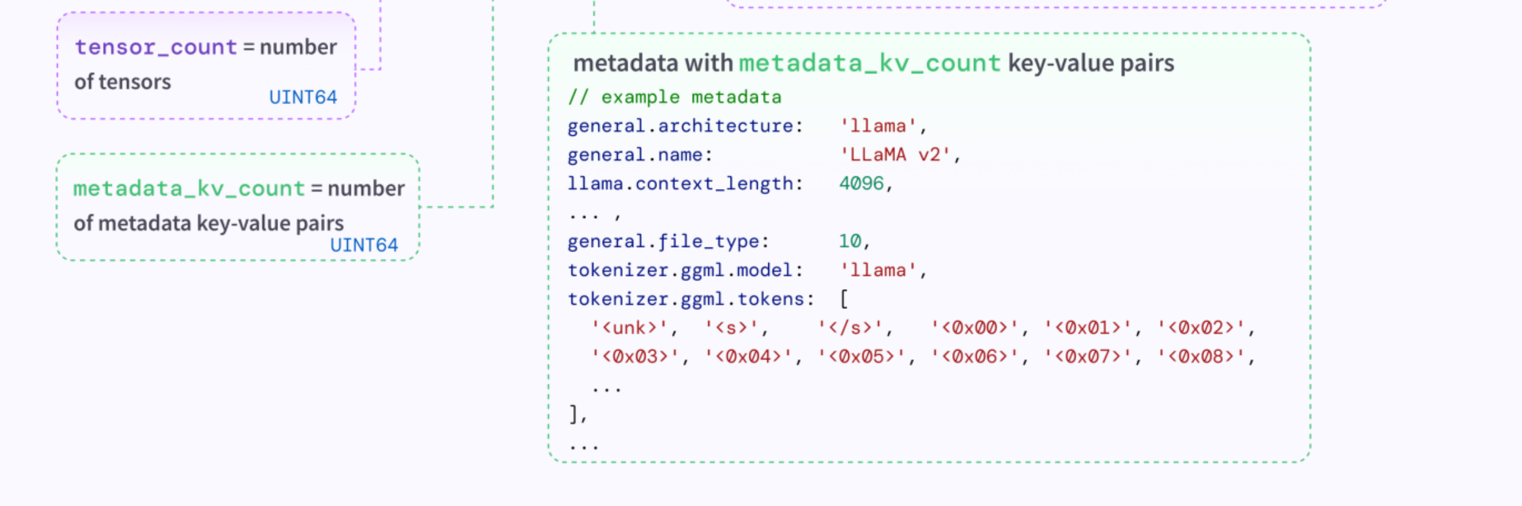
It focus on providing support for LLama and Whisper models, through llama.cpp and whisper.cpp respectively, backed by the GGML library developed by Gerganov.

An important difference compared to Safetensors is that GGUF strives to bundle everything you need to use an LLM into a single file, including the model vocabulary.

GGUF Data Format

The data format of the .gguf is a bit more complicated than .safetensors, and contains much more standardized metadata:





Let's read the header of the "Meta-Llama-3-8B.gguf" that we got from the previous section with Python:

```python
from struct import unpack

with open('Meta-Llama-3-8B.gguf', mode='rb') as f:
    # 'GGUF' magic number
    magic_number = f.read(4).decode('utf-8')
    print(f'Magic number: {magic_number}')

    # GGUF version
    version, = unpack('<i', f.read(4))
    print(f'Version: {version}')

    # number of tensors
    tensor_count, = unpack('<Q', f.read(8))
    print(f'Tensor count: {tensor_count}')

    # number of metadata key-value pairs
    metadata_kv_count, = unpack('<Q', f.read(8))
    print(f'Metadata key-value count: {metadata_kv_count}')
```

Here is the result:

```
Magic number: GGUF
Version: 3
Tensor count: 291
Metadata key-value count: 22
```

I got lazy this time and I skipped parsing the metadata fields. Also, gguf_reader.py didn't help, it looks like it's broken at the moment.

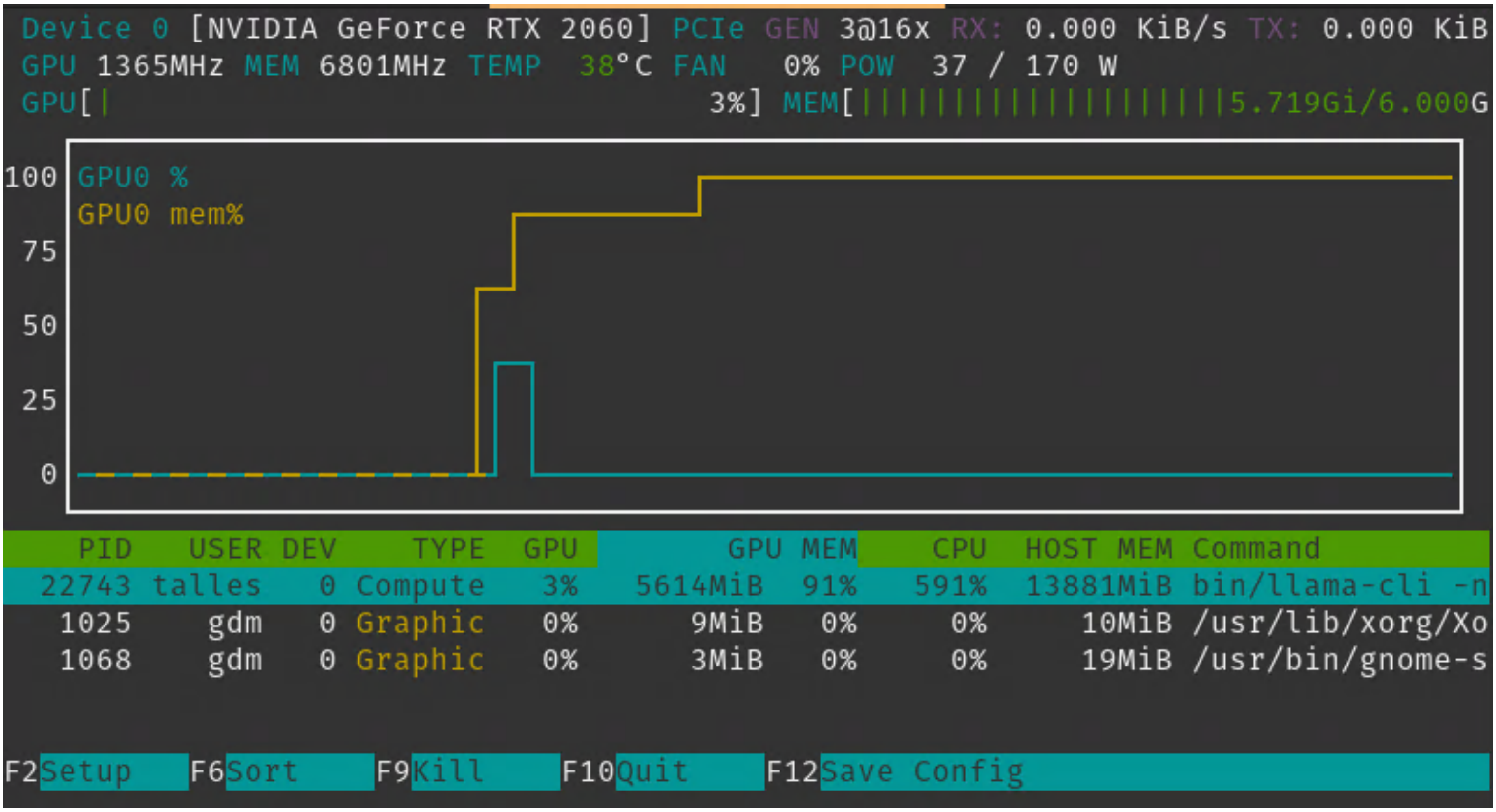
GGUF Inference (with llama.cpp)

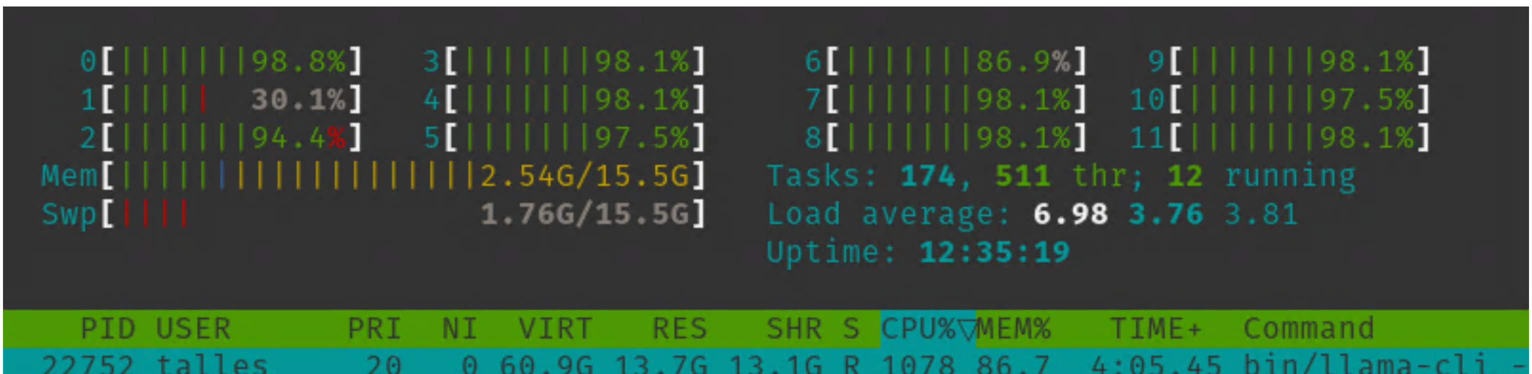
Let's get a fresh clone of llama.cpp, build it, and use the handy **"llama-cli"** to prompt our .gguf model:

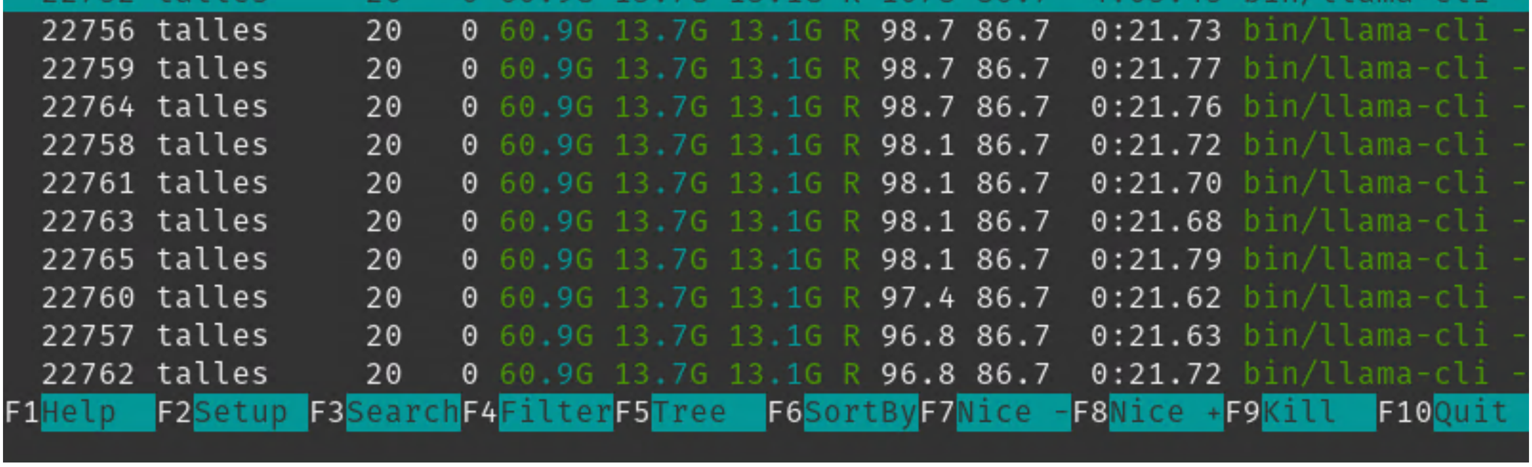
```
$ git clone https://github.com/ggerganov/llama.cpp.git
```

```
Cloning into 'llama.cpp'...
remote: Enumerating objects: 27710, done.
...

$ cd llama.cpp

$ LLAMA_CUDA=1 make -j
[  0%] Generating build details from Git
[  3%] Building CUDA object CMakeFiles/ggml.dir/ggml-cuda/concat.cu.o
...

$ bin/llama-cli -t 11 -ngl 9 -m gguf-py/Meta-Llama-3-8B.gguf -p "AMD vs NVIDIA is an interesting topic"
Log start
main: build = 3209 (95f57bb5)
main: built with cc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0 for x86_64-linux-gnu
...
AMD vs NVIDIA is an interesting topic because both are in the same field of graphics cards.
Both AMD and NVIDIA provide excellent graphics cards that will make your experience on your computer, gamir
...
```

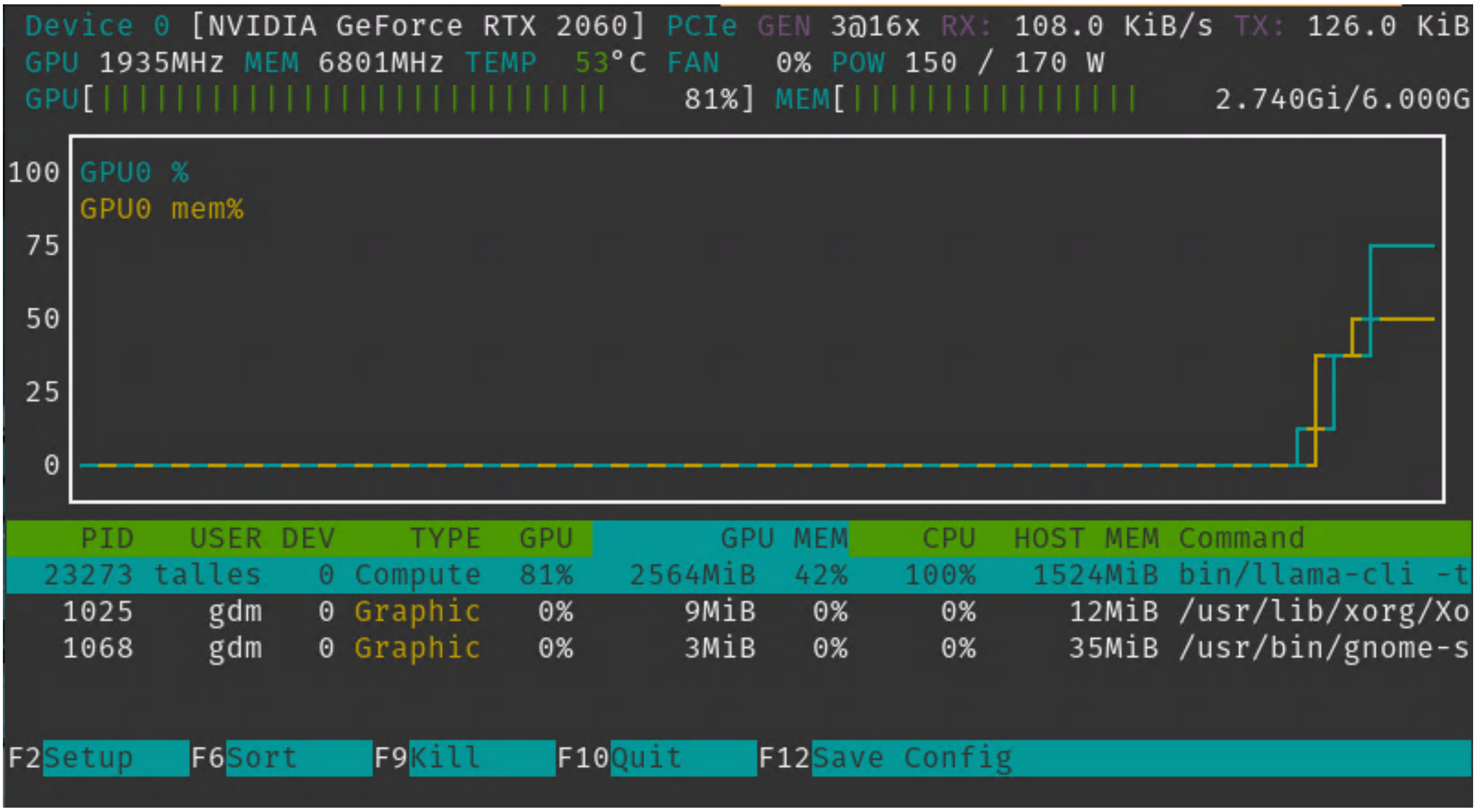
As with the previous example, I have run this on a humble RTX 2060 6GB, which is not able to host the entire Llama 3 8B model on its VRAM. The maximum I managed to fit was 9 layers with "-ngl 9". Part of the model got loaded into RAM and my CPU was used to generate the prediction. I got usable but poor token generation (~2 tokens per second):







I decided to put it to the test if my poor performance was due to the model not being able to fit entirely on my VRAM, and looks like this is the case. I got a .gguf of Gemma 2B, a smaller model, and loaded it using the same command. It fitted on VRAM and token generation was blazing fast (~130 tokens per second):



Which One to Pick?

On one hand, SafeTensors was designed with safety as a primary goal from the very beginning. Being backed by Hugging Face, it is poised to eventually become the standard. On the other hand, incredibly popular tools such as Ollama are backed by GGML and its GGUF format.

From my experience with both, it boils down to:

- Do you want to be closer to what seems to be becoming the standard (Safetensors) or do you prefer to stay closer to the GGML/llama.cpp/whisper.cpp set of tools (GGUF)?
- Is your focus to further develop pre-trained models (Safetensors) or are you mostly concerned with the ease of distribution of the model together with hyperparameters defined

by you (GGUF)?

Sources

Safetensors:

- Safetensors (Hugging Face)
- Safetensors (GitHub)
- Load safetensors (Hugging Face)
- Chat with Transformers (Hugging Face)
- Handling big models for inference (Hugging Face)
- Instantiate a big model (Hugging Face)
- Audit shows that safetensors is safe and ready to become the default (Hugging Face)

GGUF:

- GGML - AI at the edge (ggml.ai)
- llama.cpp (GitHub)
- GGUF (GitHub)
- GGUF (Hugging Face)
- ggml : unified file format #220 (GitHub)
- GGML - Large Language Models for Everyone (GitHub)